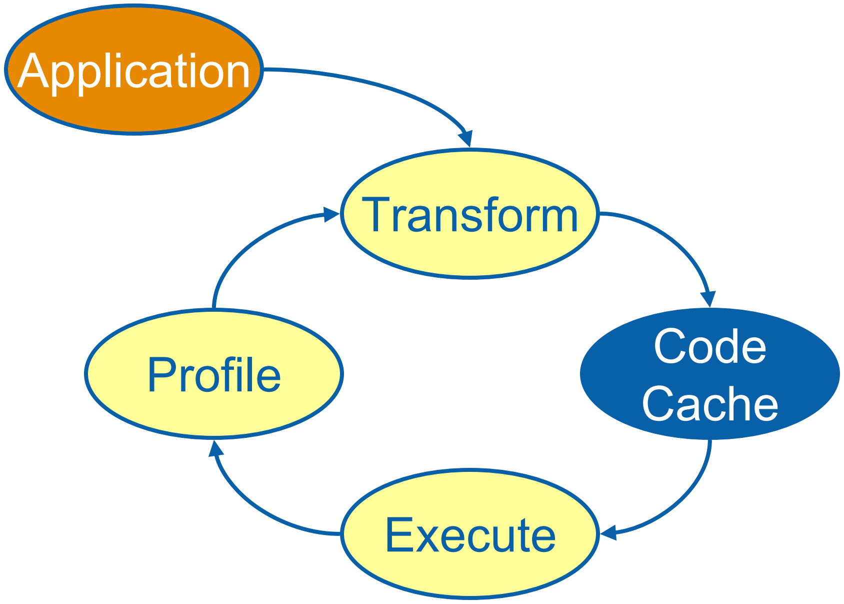
A Cross-Architectural Interface for Code Cache Manipulation

Kim Hazelwood and Robert Cohn

Software-Managed Code Caches

- Software-managed code caches store transformed code at run time to amortize overhead of dynamic optimizers
- Contain a (potentially altered) copy of application code



Code Cache Contents

Every application instruction executed is stored in the code cache (at least)

Code Regions

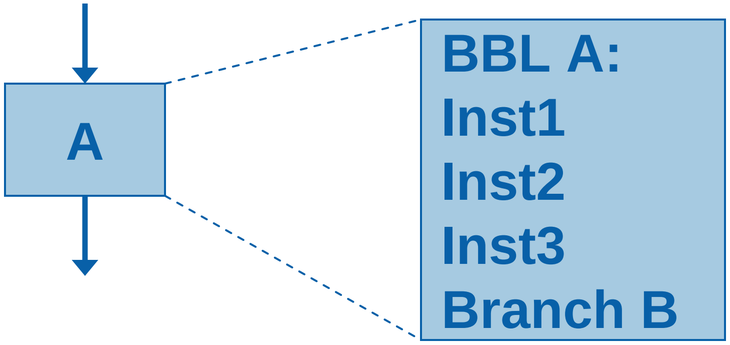
- Altered copies of application code
- Basic blocks and/or traces

Exit stubs

- Swap application⇔VM state
- Return control to the dynamic optimizer

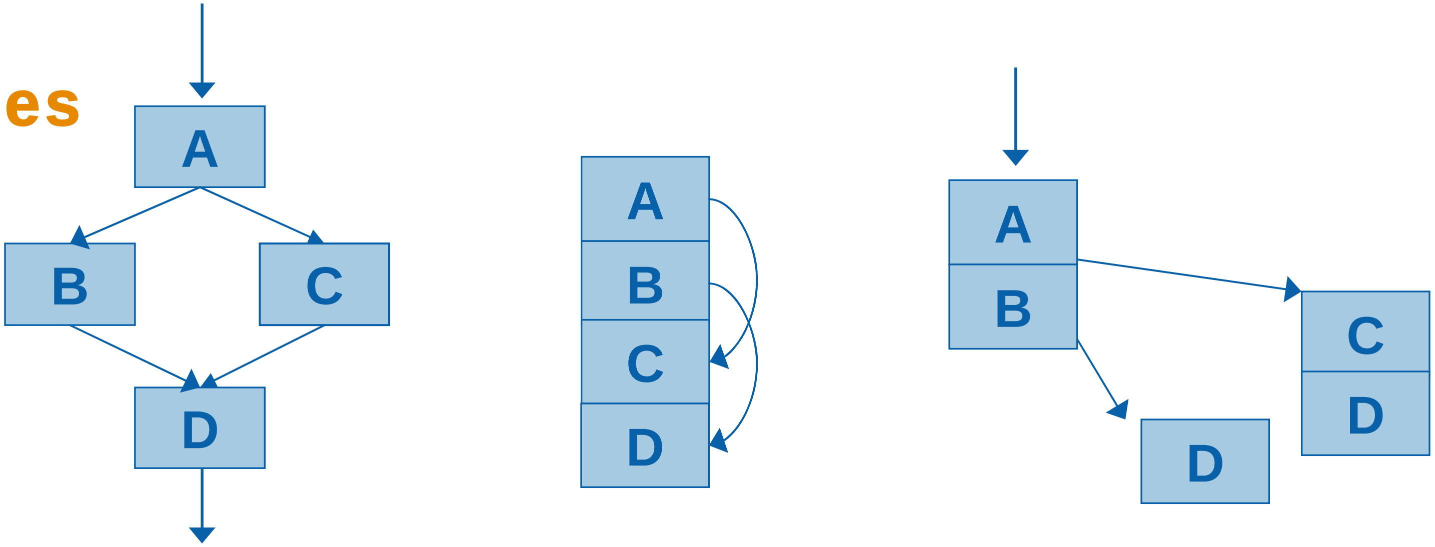
Code Regions

Basic Blocks



BBL A:
Inst1
Inst2
Inst3
Branch B

Traces



CFG In Memory Trace

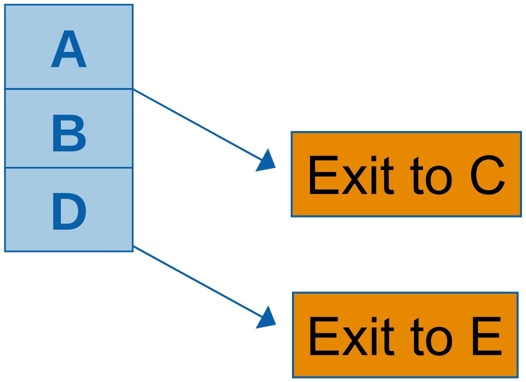
Exit Stubs

One exit stub exists for every exit from every trace or basic block

Functionality

Prepare for context switch

Return control to VM dispatch



Details

Each exit stub ≈ 3 instructions

A Goal of the Code Cache: Transparency

Pretend as though the original program is executing

Push 0x1006 on stack, then jump to 0x4000

Original Code:

```
0x1000 call  0x4000
```

Translated Code:

```
0x7000 push  0x1006
0x7006 jmp   0x8000
```

Code cache address mapping:

0x1000 → 0x7000 “caller”

0x4000 → 0x8000 “callee”

SPC

TPC

A Challenge and an Opportunity

Challenges

- Code caches hold the key to overall performance
  - ➢ Self-modifying code
  - ➢ Unloaded libraries
  - ➢ Bounded sizes

Opportunities

- Ephemeral instrumentation
- Adaptive optimizations
- Security

Interesting Research Problems, but…

- Most systems hide all evidence of code caches
- Investigations have required source code access
- Code cache implementations are often tightly coupled to the rest of the system in subtle ways

Direct code cache access can be a powerful opportunity!

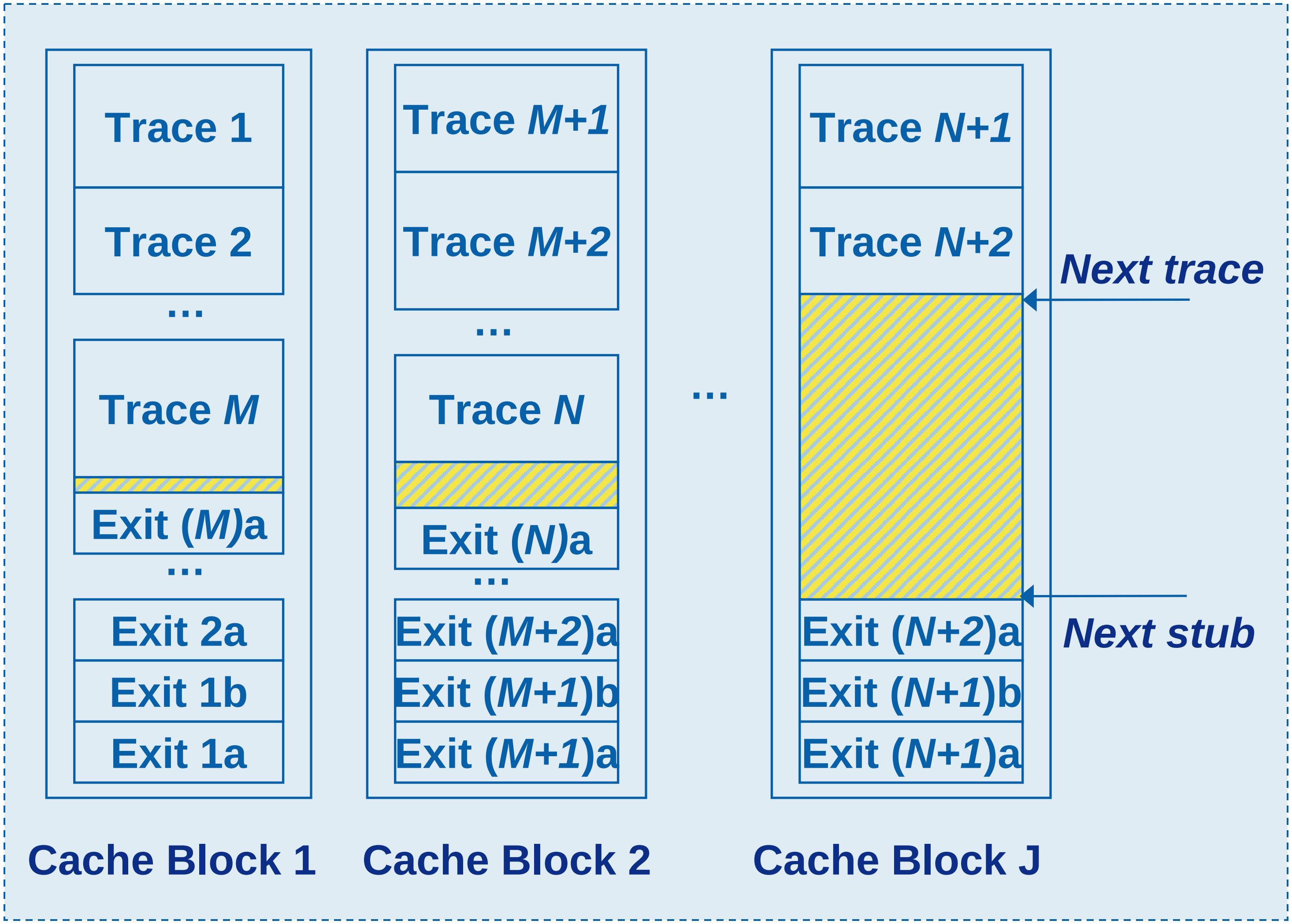
The Code Cache API

- We provide a clean, robust interface for accessing and altering code cache behavior and contents
  - ATOM-style interface
  - Rapid prototyping
- Users can
  - Investigate code cache design decisions
  - Investigate applications of binary modifiers
- Built upon the Pin dynamic instrumentation system

Building upon Pin

- A dynamic instrumentation system from Intel
- Multiple platform support
  - Four ISAs – IA32, EM64T, IPF, ARM
  - Four OSes – Linux, Windows, FreeBSD, MacOS
- Robust and stable (Pin can run itself!)
  - 12+ active developers
  - Nightly testing of 12 configurations on 25000 binaries
  - Automatic generation of user manuals
  - Large user base in academia and industry
  - Pinheads mailing list
- Seamless interaction with instrumentation interface

Pin’s Code Cache



Trace 1

Trace 2

…

Trace *M*

Exit (*M*)a

…

Exit 2a

Exit 1b

Exit 1a

Cache Block 1

Trace *M+1*

Trace *M+2*

…

Trace *N*

Exit (*N*)a

…

Exit (*M+2*)a

Exit (*M+1*)b

Exit (*M+1*)a

Cache Block 2

…

Trace *N+1*

Trace *N+2*

Next trace

Next stub

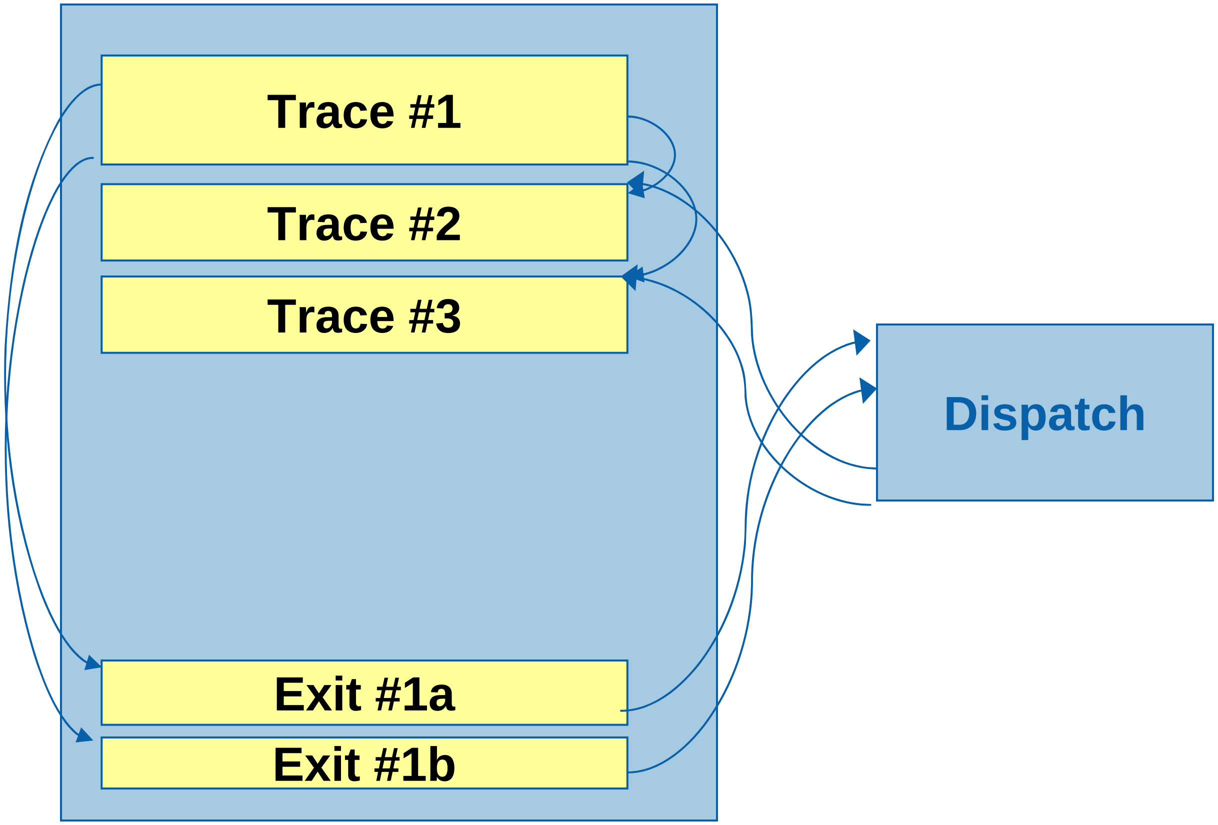
Exit (*N+2*)a

Exit (*N+1*)b

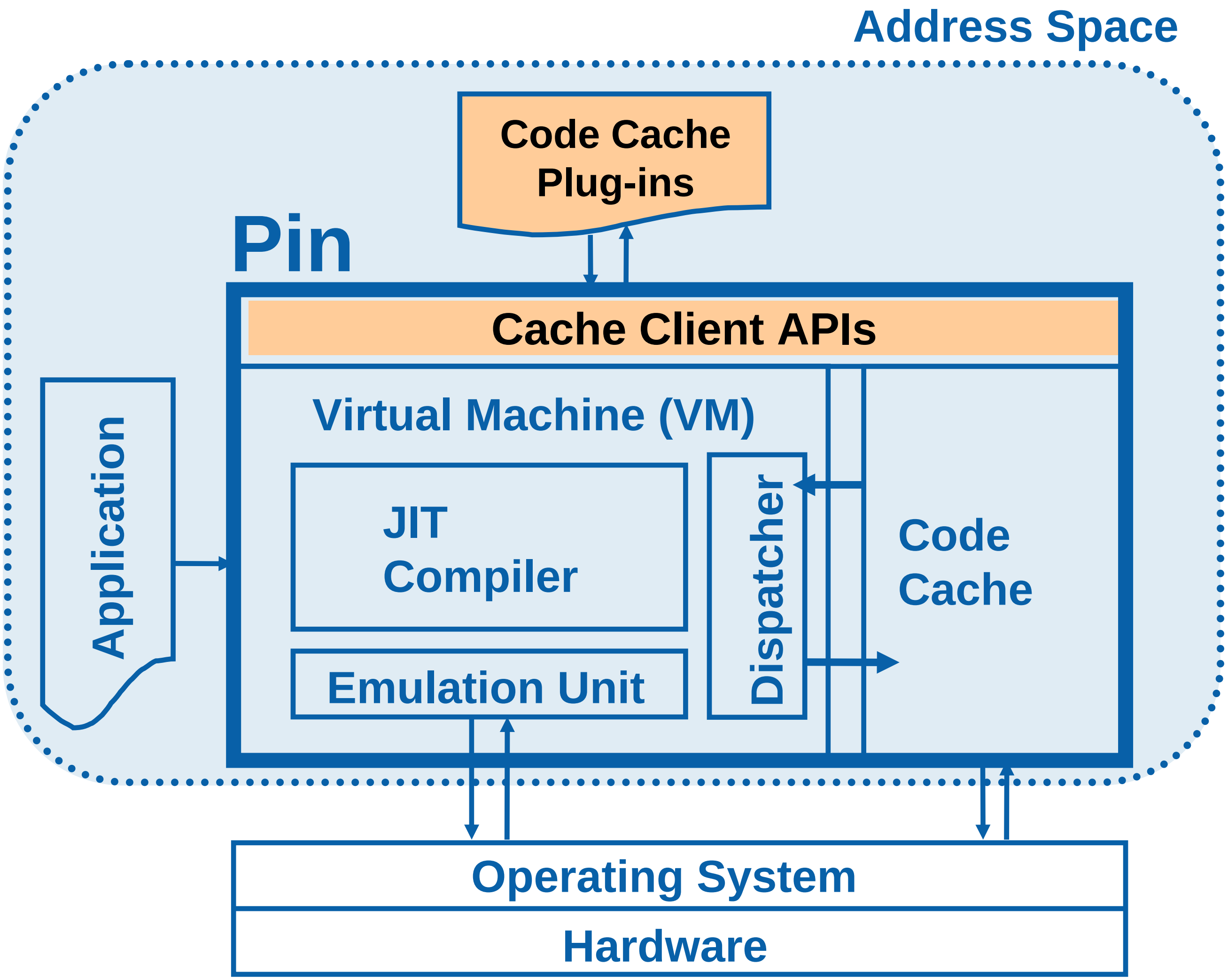
Exit (*N+1*)a

Cache Block J

Cache Linking



Cache Client Interface



Code Cache API

Callbacks – Events that trigger calls to user functions

- CacheIsFull, TraceInserted, EnteringCache, TraceLinked, BlockCreated, …

Actions – Events a user can invoke via instrumentation

- ChangeCacheSize(Sz), FlushCache(), FlushBlock(Id), InvalidateTrace(SPC)…

Lookups – Returns IDs, mappings, handles for traces, exit stubs, cache blocks, …

Statistics – Live summary of cache contents

- MemoryUsed, FlushCount, TracesInCache, CacheBlockCount, …

Our Design Goals

- Ease of use
- Comparable performance to a direct implementation

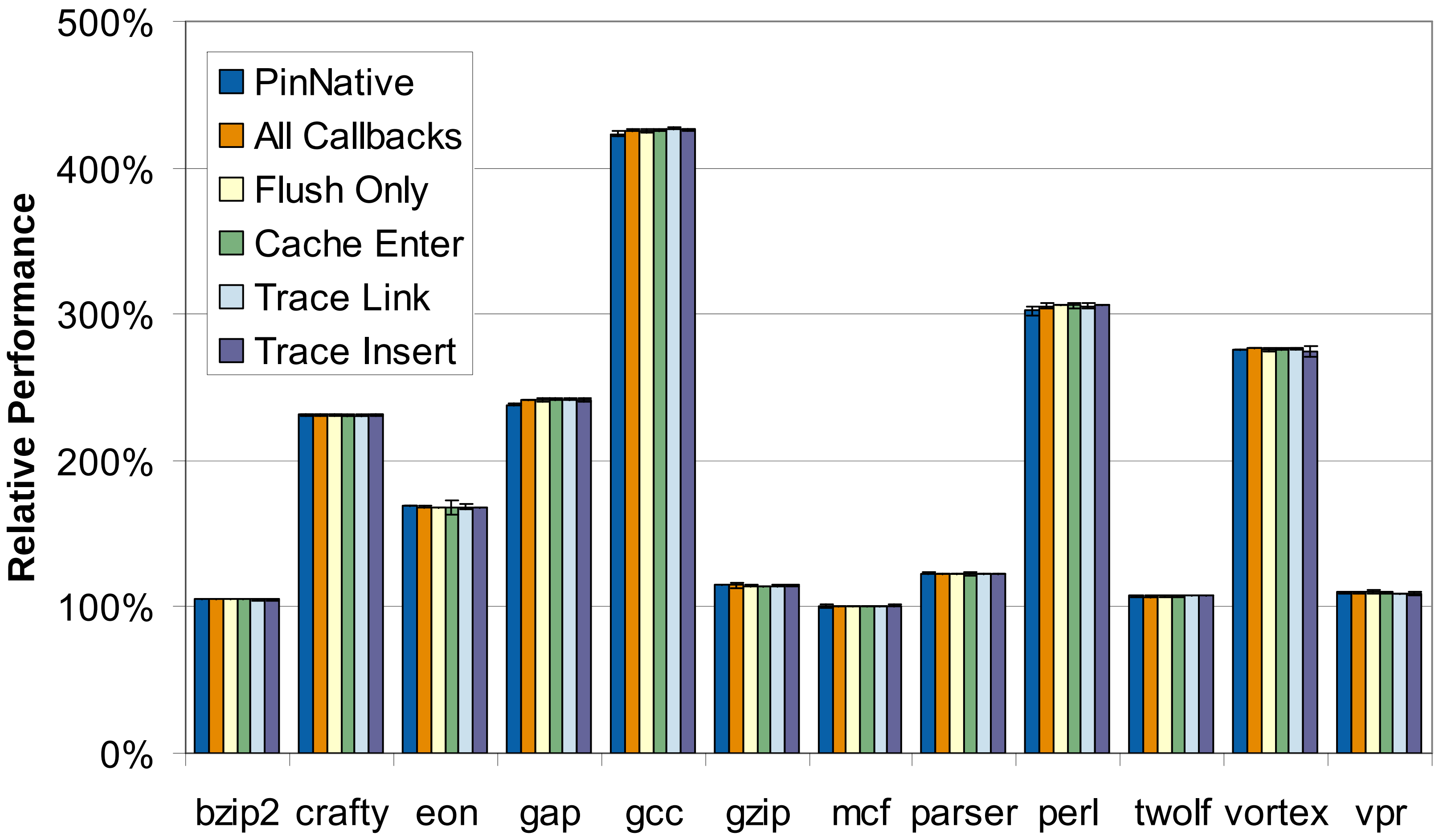
Major instrumentation overhead source

- “State switch” between executing application and instrumentation code

Fundamental difference

- Nearly all callbacks occur from the VM (no state switch)
- All others would incur the same state switch overhead in a direct implementation…

Overhead of Empty Routines



Code Cache API Utility

Code Cache Design

- Cache replacement investigations
- Graphical visualization
- Architectural comparisons
  - IA-32, EM64T, Itanium, XScale

Code Cache Applications

- Optimization algorithms
- Security algorithms

Cache Replacement

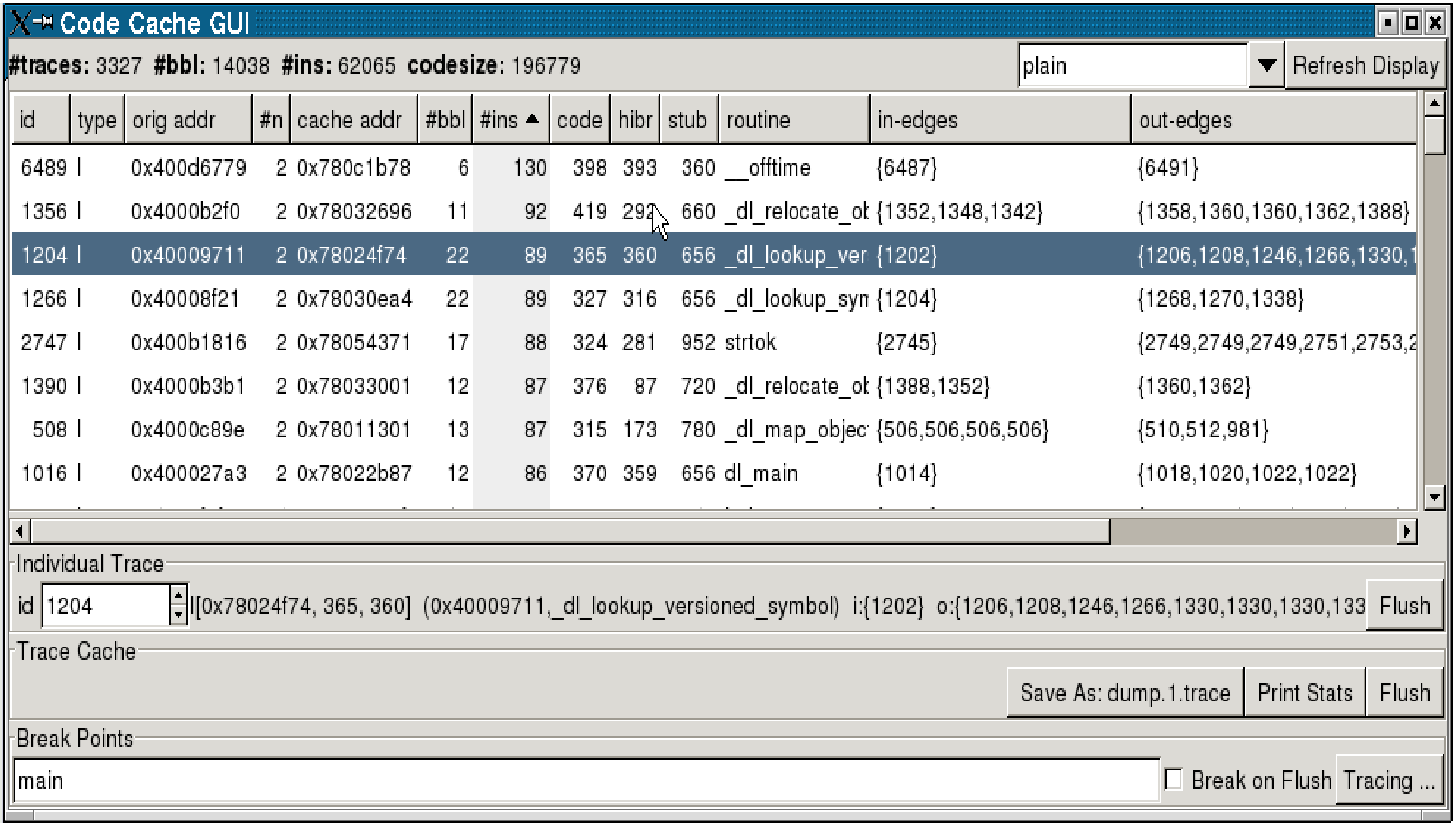
```
void main(int argc, char **argv) {
  PIN_Init(argc,argv);
  CODECACHE_CacheIsFull(FlushOnFull);
  PIN_StartProgram(); //Never returns
}

void FlushOnFull() {
  CODECACHE_FlushCache();
  cout << “SWOOSH!” << endl;
}
```

Eviction Granularities
- Entire Cache
- One Cache Block
- One Trace
- Address Range

```
% pin –cache_size 40960 –t flusher -- /bin/ls
SWOOSH!
SWOOSH!
<output of /bin/ls>
```

A Graphical Front-End



Code Cache GUI

#traces: 3327 **#bbl:** 14038 **#ins:** 62065 **codesize:** 196779

id	type	orig addr	#n	cache addr	#bbl	#ins ▲	code	hibr	stub	routine	in-edges	out-edges
6489	l	0x400d6779	2	0x780c1b78	6	130	398	393	360	__offtime	{6487}	{6491}
1356	l	0x4000b2f0	2	0x78032696	11	92	419	292	660	_dl_relocate_ol	{1352,1348,1342}	{1358,1360,1360,1362,1388}
1204	l	0x40009711	2	0x78024f74	22	89	365	360	656	_dl_lookup_ver	{1202}	{1206,1208,1246,1266,1330,1
1266	l	0x40008f21	2	0x78030ea4	22	89	327	316	656	_dl_lookup_syn	{1204}	{1268,1270,1338}
2747	l	0x400b1816	2	0x78054371	17	88	324	281	952	strtok	{2745}	{2749,2749,2749,2751,2753,2
1390	l	0x4000b3b1	2	0x78033001	12	87	376	87	720	_dl_relocate_ol	{1388,1352}	{1360,1362}
508	l	0x4000c89e	2	0x78011301	13	87	315	173	780	_dl_map_objec	{506,506,506,506}	{510,512,981}
1016	l	0x400027a3	2	0x78022b87	12	86	370	359	656	dl_main	{1014}	{1018,1020,1022,1022}

Individual Trace

id 1204 l[0x78024f74, 365, 360] (0x40009711,_dl_lookup_versioned_symbol) i:{1202} o:{1206,1208,1246,1266,1330,1330,1330,133

Trace Cache

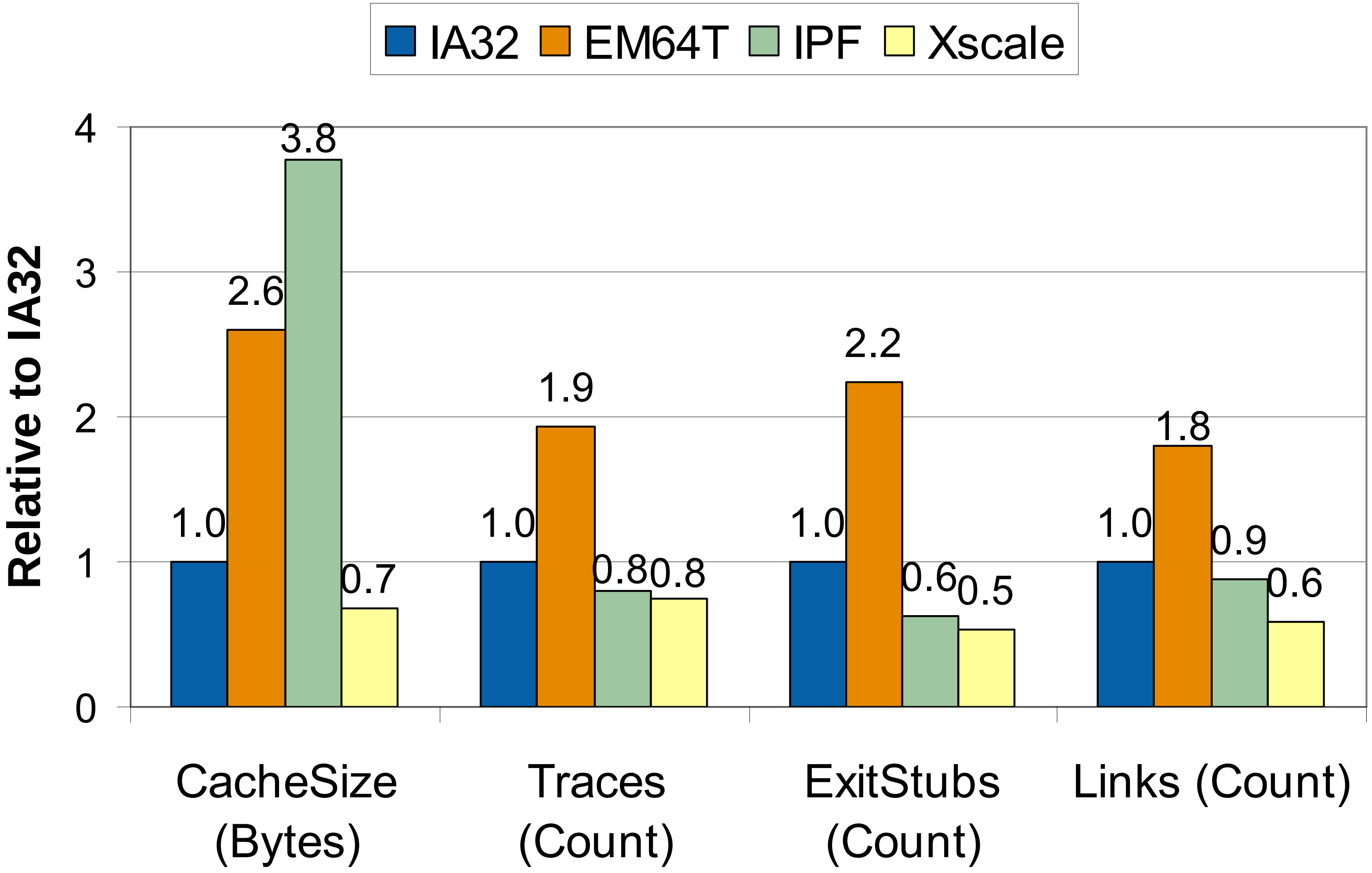
Break Points

main

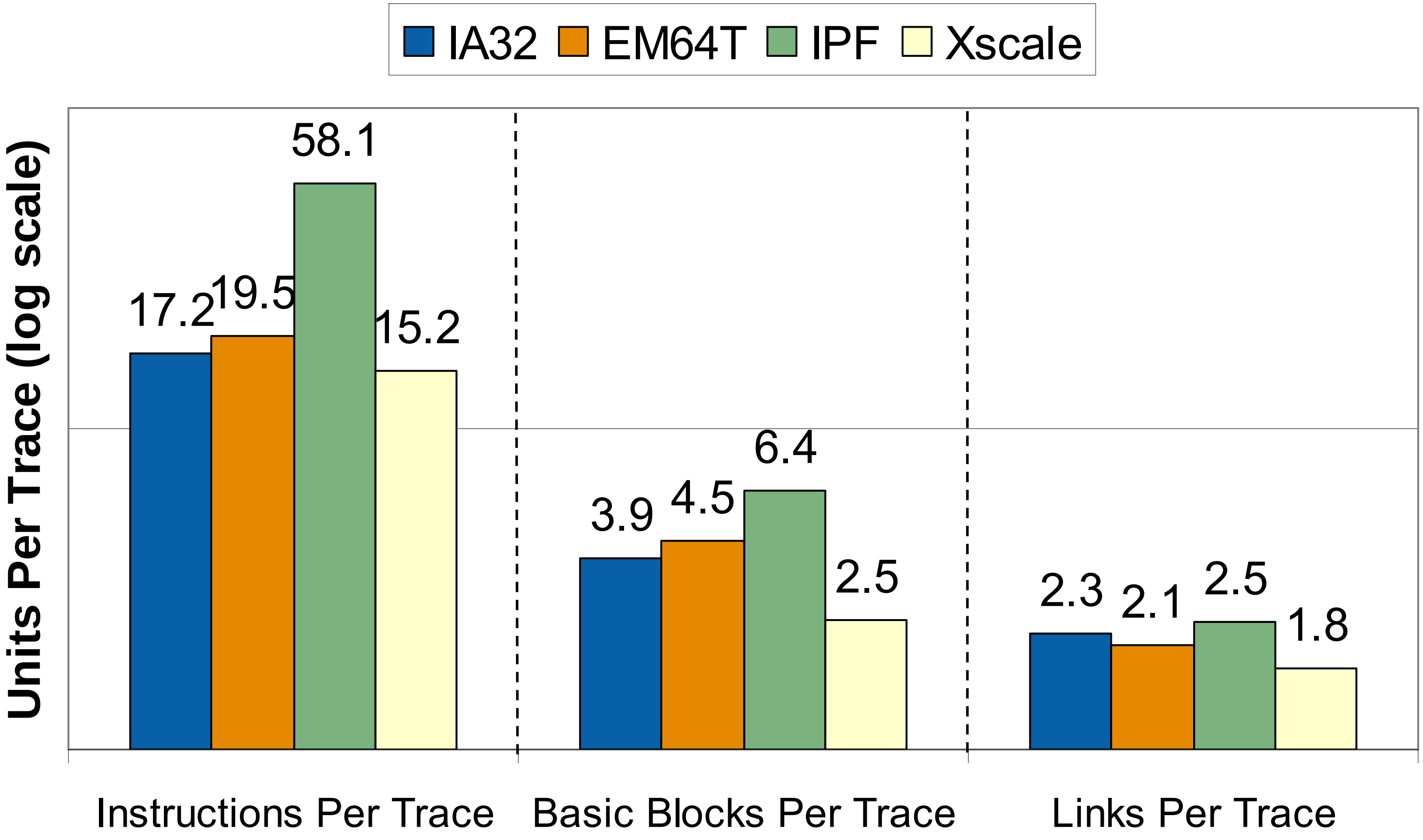
Design Challenge: ISA Idiosyncrasies

	ARM	IA-32/EM64T	IPF
Type	RISC	CISC	VLIW
Instruction	Fixed length	Variable length, prefixes	Bundled
Memory Instruction	LD/ST	Any, implicit	LD/ST
Memory op size	Fixed	Variable length	Fixed
Addressing modes	Pre/post/iprel increment	Index/offset/ scale/iprel	Post
Predication	Cond. codes	None	Predicate regs
Parameters	Registers	Stack/registers	Stacked registers

Architectural Comparisons



Architectural Comparisons (2)



Design Challenge: Self-Modifying Code

The problem

Code cache must detect SMC and invalidate corresponding cached traces

Solutions

Many proposed … but source code is usually necessary to investigate solutions

Self-Modifying Code Handler

```
void main (int argc, char **argv) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(InsertSmcCheck,0);
   PIN_StartProgram(); // Never returns
}
void InsertSmcCheck () {
    . . . (variable declarations) . . .
    memcpy(traceCopyAddr, traceAddr, traceSize);
    TRACE_InsertCall(trace, IPOINT_BEFORE, (AFUNPTR)DoSmcCheck,
         IARG_PTR, traceAddr, IARG_PTR, traceCopyAddr,
         IARG_UINT32, traceSize, IARG_CONTEXT, IARG_END);
}
void DoSmcCheck (VOID* traceAddr, VOID *traceCopyAddr,
         USIZE traceSize, CONTEXT* ctxP) {
   if (memcmp(traceAddr, traceCopyAddr, traceSize) != 0) {
         CODECACHE_InvalidateTrace((ADDRINT)traceAddr);
         PIN_ExecuteAt(ctxP);
   }
}
```

(Written by Alex Skaletsky)

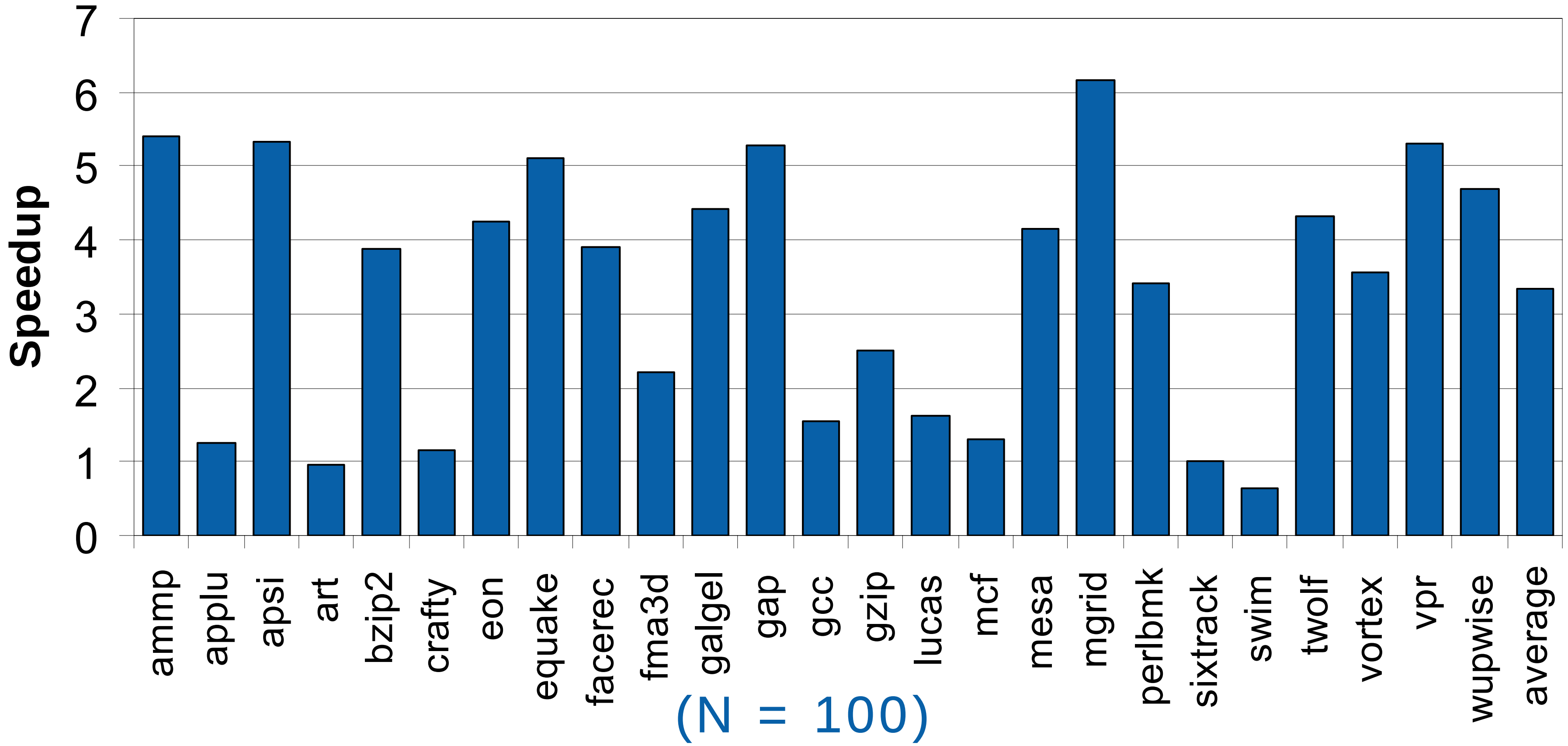
Adaptive Code Optimizations

Many reasons to selectively invalidate cached traces

- Ephemeral profiling
- Phase-based optimizations
- Adaptive algorithms

Two-Phase Instrumentation

- Memory reference instrumentation can be costly
- Can invalidate instrumented code after *N* executions



(N = 100)

Plug-In Tools Shipped with our API

CacheSimulator – Exercises most of the API

CacheFlusher – Performs a full cache flush

CacheDoubler – Doubles cache size when full

LinkUnlink – Watches all link activity

BBTest – User-defined trace sizing

TraceInvalidator – Invalidates hot traces

SMCHandler – Stores a copy of program instructions and invalidates stale code

. . . and several more . . .

Summary

- Low overhead but highly functional interface to Pin’s code cache
- Enables introspection as well as adjustment of cache policies and contents
- One API → four ISAs → four OSes
- Works seamlessly with Pin’s instrumentation API

- Download it today!

http://rogue.colorado.edu/pin